School of Computation, Information and Technology
Technische Universität München

Lehrstuhl für
Rechnerarchitektur & Parallele Systeme
Prof. Dr. Martin Schulz
Dominic Prinz
Jakob Schäffeler

Lehrstuhl für
Design Automation
Prof. Dr.-Ing. Robert Wille
Stefan Engels

Einführung in die Rechnerarchitektur

Wintersemester 2024/2025

Übungsblatt 8: RISC-V Prozessor 09.12.2024 – 13.12.2024

1 Maschinensprache

- Übersetze das folgende RISC-V Assembly Programm in RISC-V Maschinensprache

```
add s7, s8, s9
sll t1, t2, t3
srli s3, s1, 14
sw s9, 16(t4)
```

16 8 4 2 1
1 0 1 1 1

add: R-Type

funct7	rs2	rs1	f3	rd	opcode
0000000	11001	11000	000	10111	0110011
0	1	9	c	0	b	b	3

0x 0 1 9 c 0 b b 3

sll: R-Type

funct7	rs2	rs1	f3	rd	opcode
0000000	11100	00111	001	00110	0110011
0	1	c	7	3	3

0x 0 1 c 7 3 3

srli: I-Type

Immediate 11:0	rs1	funct3	rd	op
000000001110	01001	101	10011	0010011
0	0	e	4	d	9	9	3

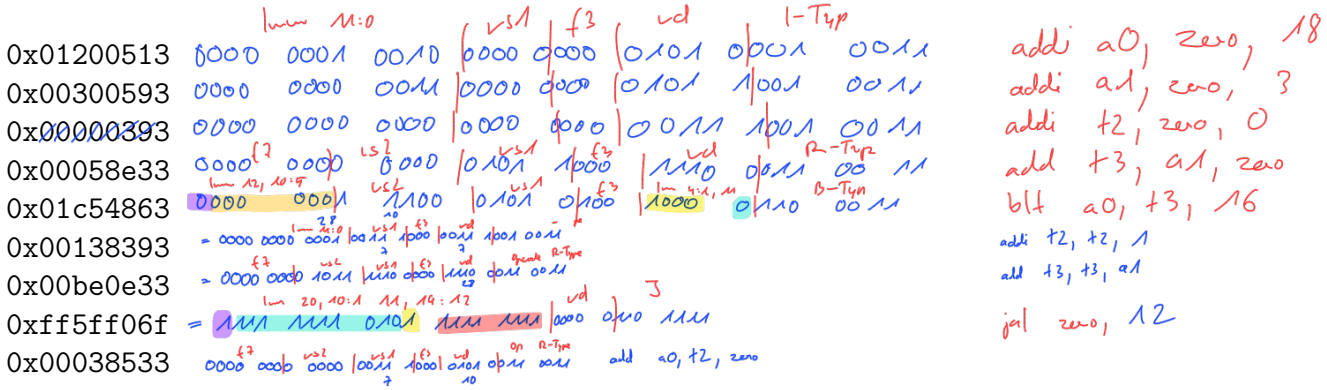
= 0x00e4d993

sw S-Type

Immediate 11:5	rs2	rs1	funct3	imm 4:0	op
0000000	11001	11101	010	10000	0100011
0	1	9	e	a	8	2	3

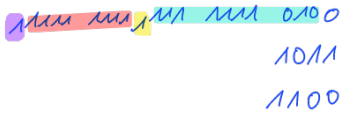
= 0x19ea823

- Konvertiere den folgenden RISCV Maschinencode zurück in RISC-V Assembly:



```
0x01200513   addi a0, zero, 18
0x00300593   addi a1, zero, 3
0x00000393   addi t2, zero, 0
0x00058e33   add  t3, a1, zero
0x01c54863   blt  a0, t3, 16
0x00138393   addi t2, t2, 1
0x00be0e33   add  t3, t3, a1
0xff5ff06f   jal  zero, 12
0x00038533   add  a0, t2, zero
```

- Erkläre was das Programm aus der vorherigen Aufgabe berechnet.



```
addi a0, zero, 18
addi a1, zero, 3
addi t2, zero, 0
add  t3, a1, zero
loop: blt  a0, t3, end
      addi t2, t2, 1
      add  t3, t3, a1
      jal  zero, loop
end:  add  a0, t2, zero
```

Division

$\frac{18}{2}$

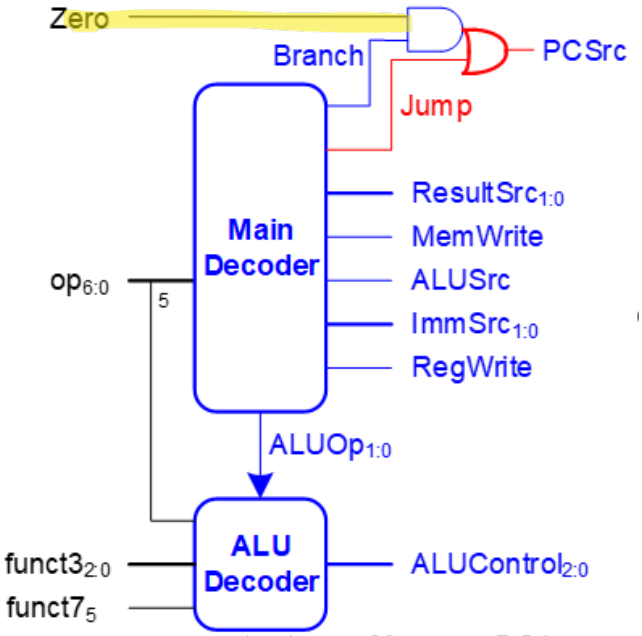
2 Fehlerhafte Kontrollsignale

Angenommen der Prozessor in Abbildung 2 hat fehlerhafte Kontrollsignale, deren Wert konstant auf einen bestimmten Wert gelegt ist. Die folgende Tabelle listet Kontrollsignale und deren konstanten Wert auf. Welche der Befehle, die in der Vorlesung vorgestellt wurden, funktionieren nicht mehr unter dieser fehlerhaften Belegung?

Kontrollsignal	Konstante Belegung	Fehlerhafte Befehle
ALUControl	000 (ADD)	R-Typ (außer ADD), I-Typ (außer addi, lw), BEQ
PCSrc	0	BEQ, JAL
ResultSrc	01	R-Typ, I-Typ (außer lw), JAL

3 Erweiterung: BNE

Das Steuerwerk des Prozessors sieht im genaueren wie folgt aus:



Insbesondere ist die Logik des `PCSrc` Signals genauer abgebildet: PCSrc wird auf 1 gesetzt wenn Jump auf 1 (bei JAL) ist oder ein Branch gemacht wird und das `Zero` Signal 1 liefert (bei BEQ).

Erweitere das Steuerwerk und gegebenfalls das Prozessorschaltbild so, dass auch ein BNE (Branch Not Equal) Befehl durchgeführt werden kann. Der BNE Befehl vergleicht zwei Quellregister und addiert einen relativen Offset auf den PC falls die Register *nicht* gleich sind. Der Maschinencode des BNE ist in Abbildung 3 abgebildet.

4 Kontrollsignalbelegung

Gebe die Belegung der Kontrollsignale und den Befehlstyp für folgende Befehle an. Für die ALUControl reicht der Name der ALU-Instruktion (ADD, SUB, AND, ...). Gebe Signale bei denen die Belegung keinen Einfluss auf die Funktionalität hat explizit mit *Don't Care* ('x') an.

Befehl	Befehlstype	Branch	ResultSrc	MemWrite	ALUControl	ALUSrc	ImmSrc	RegWrite
OR	R	0	01	0	OR	0	XX	1
BNE	B	1	XX	0	SUB	0	10	0
XORI	I	0	00	0	XOR	1	00	1
SW	S	0	XX	1	ADD	1	01	0

5 Prozessorerweiterung (Hausaufgabe)

Bearbeitung und Abgabe der Hausaufgabe 8 auf `https://artemis.in.tum.de/courses/401` bis **Sonntag, den 15.12.2024, 23:59 Uhr**.

Ziel dieser Übung ist es den in der Vorlesung vorgestellten Prozessor um zwei Befehle zu erweitern und in Digital zu implementieren. Die in der Vorlesung und Zentralübung vorgestellte Funktionalität ist bereits in der Vorlage vorgegeben.

Bevor du mit der Implementierung beginnst, überlege dir zuerst am Schaltbild in Abbildung 2 wie die Kontrollsignale gesetzt werden müssen und welche zusätzliche Komponenten oder Signale eventuell benötigt werden.

Da die Funktionalität des Prozessors getestet werden soll, sind die Testbenches in dieser Übung teilweise Assemblyprogramme. Diese können über Rechtsklick -> `Daten: Bearbeiten` -> `Datei` -> `Laden` direkt ins ROM *InstrMem* in der Datei `SingleCycleRiscV.dig` geladen werden oder selbst geschrieben werden. Auf Artemis sind im Template bereits kleine Programme zum Testen[1] in den `.hex` Dateien vorhanden.

Die folgenden Teilaufgaben können unabhängig voneinander gelöst werden:

a) Erweitere den gegebenen Prozessor, sodass dieser die Instruktion *xor* unterstützt. In dieser Teilaufgabe dürfen nur die Dateien `ALU.dig` und `ALUDecoder.dig` angepasst werden. Änderungen in anderen Dateien werden vom Tester ignoriert. Die ALUControl der *xor*-Funktion soll `100` sein. Die Ein- und Ausgänge dürfen nicht verändert werden.

   Hinweis zum Decoder: Überlege dir zunächst welchen eindeutigen(!) Wert ALUOp *und* funct3 *haben und verbinde eine Konstante an den richtigen Eingang eines Multiplexers.*

b) Erweitere den gegebenen Prozessor, sodass dieser die Instruktion *jalr* unterstützt. In dieser Teilaufgabe dürfen alle Dateien inklusive des Prozessorschaltbildes geändert werden. Die bestehenden Ein- und Ausgänge dürfen nicht verändert werden und es dürfen keine zusätzlichen Eingänge hinzugefügt werden. Es ist jedoch (nur in Teil b) explizit erlaubt einzelnen Subcircuits zusätzliche Ausgänge hinzuzufügen.

   Hinweis: Die in Teilaufgabe a angepassten Schaltkreise müssen für diese Teilaufgabe nicht verändert werden. Versuche bei der Implementierung möglichst viele Komponenten wiederzuverwenden.

Hinweis: Es dürfen sämtliche Digital-Komponenten aus den Bereichen „Logisch“, „IO“, „Leitungen“, „Multiplexer“, „FlipFlops“, „Speicher“ und „Arithmetik“ verwendet werden.

[1] Um zu sehen ob der Prozessor korrekt ist, überprüfe ob am Ende des Programms die richtigen Ergebnisse im Datenspeicher stehen. Diese werden u.a. im Messwertgraph angezeigt.

6 Referenzmaterial

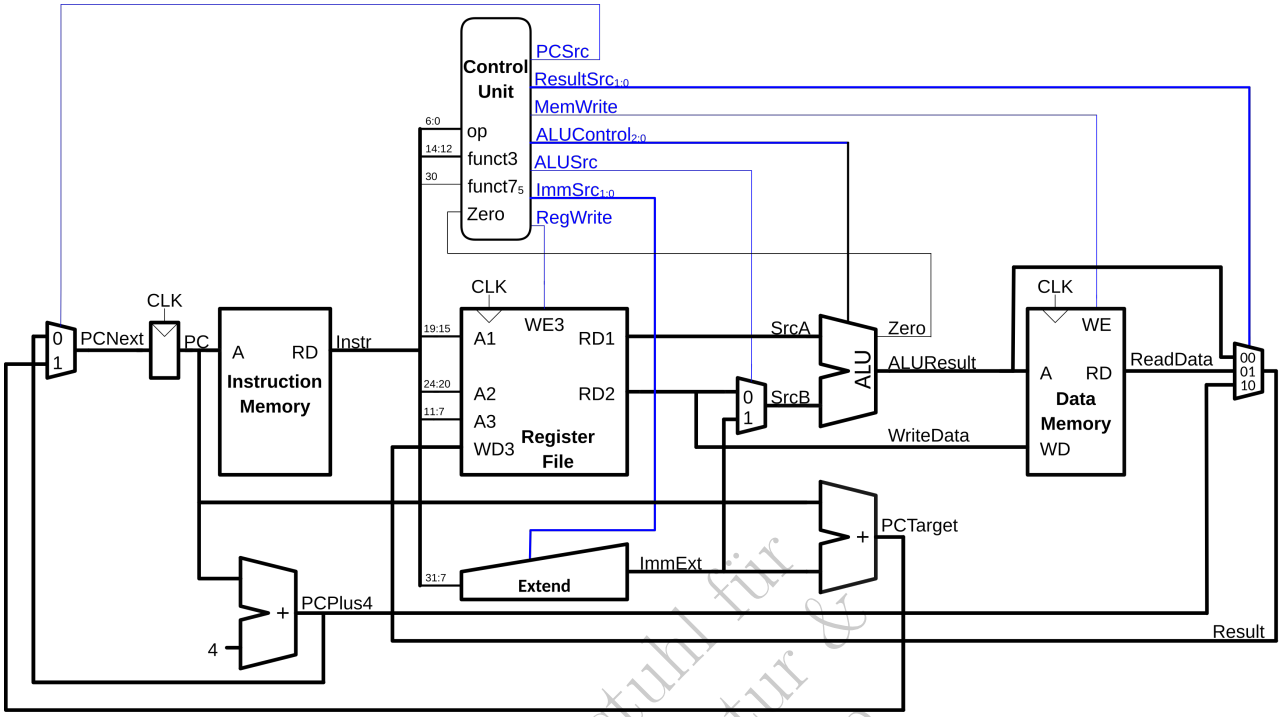


Abbildung 1: Schaltbild des Single Cycle RISC-V Prozessors

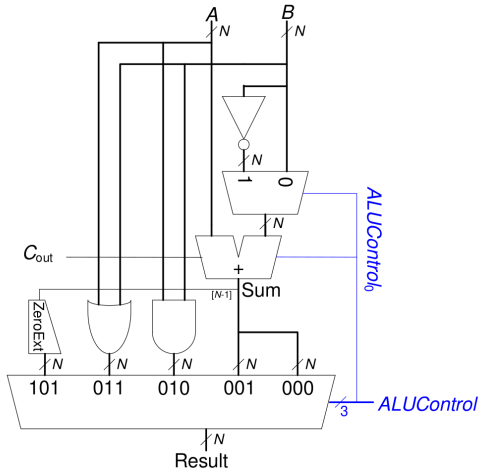


Abbildung 2: ALU des Single Cycle RISC-V Prozessors

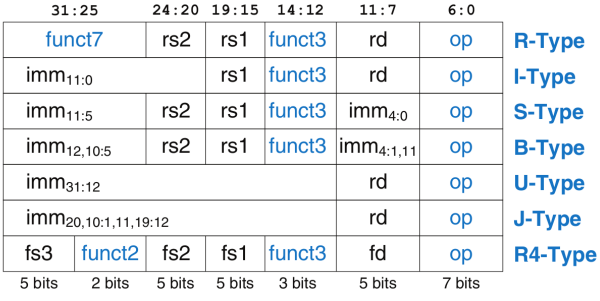
op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0= rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0= rs2`
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 − rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs24:0`
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

*Encoded in $instr_{31:25}$, the upper seven bits of the immediate field

Abbildung 3: RISC-V 32-bit Integerbefehl

$ImmSrc_{1:0}$	ImmExt	Instr. Type
00	{{20{instr[31]}}, instr[31:20]}	I-Type
01	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type
10	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type
11	{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}	J-Type

(a) Extend Unit



(b) RISC-V 32-bit Befehlsformat

Name	Nummer	Beschreibung
zero	x0	Konstante 0 (kann nicht überschrieben werden)
ra	x1	Rücksprungadresse
sp	x2	Stack-Zeiger
gp	x3	Zeiger auf globale Daten
tp	x4	Zeiger auf thread-lokale Daten
t0-2	x5 – 7	Temporäre Register*
s0/fp	x8	Stack-Frame-Zeiger (optional)**
s1	x9	Gesicherte Register**
a0-1	x10 – 11	Funktionsargumente/Rückgabewerte
a2-7	x12 – 17	Funktionsargumente
s2-11	x18 – 27	Gesicherte Register**
t3-6	x28 – 31	Temporäre Register*

Abbildung 5: Registertabelle

